下载APP

01 | 认识容器：容器的基本操作和实现原理

2020-11-16 李程远

容器实战高手课 进入课程 >

讲述：李程远
时长 24:04 大小 22.04M

你好，我是程远。作为一名工程师，我猜在过去的几年时间里，你肯定用过或者听人提起过容器（Container）。

说实话，容器这东西一点都不复杂，如果你只是想用的话，那跟着Docker 官网的说明，应该十来分钟就能搞定。

简单来说，它就是个小工具，可以把你想跑的程序，库文件啊，配置文件都一起“打包”。

然后，我们在任何一个计算机的节点上，都可以使用这个打好的包。有了容器，一个命令就能把你想跑的程序跑起来，做到了**一次打包，就可以到处使用。**

今天是咱们整个课程的第一讲，我想和你来聊聊容器背后的实现机制。

当然，空讲原理也没什么感觉，所以我还是会先带着你启动一个容器玩玩，然后咱们再一起来探讨容器里面的两大关键技术—— Namespace 和 Cgroups。基本上理解了这两个概念，你就能彻底搞懂容器的核心原理了。

做个镜像

话不多说，咱们就先动手玩一玩。启动容器的工具有很多，在这里我们还是使用 Docker 这个最常用的容器管理工具。

如果你之前根本没用过 Docker 的话，那我建议你先去官网看看文档，一些基础的介绍我就不讲了，那些内容你随便在网上一搜就能找到。

安装完 Docker 之后，咱们先来用下面的命令运行一个 httpd 服务。

```
1 # docker run -d centos/httpd:latest
```

复制代码

这命令也很简单，run 的意思就是要启动一个容器， -d 参数里 d 是 Daemon 的首字母，也就是让容器在后台运行。

最后一个参数 centos/httpd:latest 指定了具体要启动哪一个镜像，比如这里咱们启动的是 centos/httpd 这个镜像的 latest 版本。

镜像是 Docker 公司的创举，也是一个伟大的发明。你想想，在没有容器之前，你想安装 httpd 的话，会怎么做？是不是得运行一连串的命令？甚至不同的系统上操作方法也不一样？

但你看，有了容器之后，你只要运行一条命令就搞定了。其实所有的玄机都在这个镜像里面。

镜像这么神奇，那它到底是怎么一回事呢？其实，镜像就是一个特殊的文件系统，

它提供了容器中程序执行需要的所有文件。具体来说，就是应用程序想启动，需要三类文件：相关的程序可执行文件、库文件和配置文件，这三类文件都被容器打包做好了。

这样，在容器运行的时候就不再依赖宿主机上的文件操作系统类型和配置了，做到了想在哪个节点上运行，就可以在哪个节点上立刻运行。

那么我们怎么来做一个容器镜像呢？

刚才的例子里，我们用的 `centos/httpd:latest` 这个镜像是 **Docker 镜像库**里直接提供的。当然，我们也可以自己做一个提供 httpd 服务的容器镜像，这里仍然可以用 Docker 这个工具来自定义镜像。

Docker 为用户自己定义镜像提供了一个叫做 Dockerfile 的文件，在这个 Dockerfile 文件里，你可以设定自己镜像的创建步骤。

如果我们自己来做一个 httpd 的镜像也不难，举个例子，我们可以一起来写一个 Dockerfile，体会一下整个过程。用 Dockerfile build image 的 Dockerfile 和对应的目录我放在这里了。

操作之前，我们首先要理解这个 Dockerfile 做了什么，其实它很简单，只有下面这 5 行：

复制代码

```
# cat Dockerfile
FROM centos:8.1.1911
RUN yum install -y httpd
COPY file1 /var/www/html/
ADD  file2.tar.gz /var/www/html/
CMD ["/sbin/httpd", "-D", "FOREGROUND"]
```

我们看下它做了哪几件事：在一个 centos 的基准镜像上安装好 httpd 的包，然后在 httpd 提供文件服务的配置目录下，把需要对外提供的文件 file1 和 file2 拷贝过去，最后指定容器启动以后，需要自动启动的 httpd 服务。

有了这个镜像，我们希望容器启动后，就运行这个 httpd 服务，让用户可以下载 file1 还有 file2 这两个文件。

我们具体来看这个 Dockerfile 的每一行，第一个大写的词都是 Dockerfile 专门定义的指令，也就是 FROM、RUN、COPY、ADD、CMD，这些指令都很基础，所以我们不做详细解释了，你可以参考 Dockerfile 的官方文档。

我们写完这个 Dockerfile 之后，想要让它变成一个镜像，还需要执行一下 docker build 命令。

下面这个命令中 -f ./Dockerfile 指定 Dockerfile 文件，-t registry/httpd:v1 指定了生成出来的镜像名，它的格式是"name:tag"，这个镜像名也是后面启动容器需要用到的。

复制代码

```
1 # docker build -t registry/httpd:v1 -f ./Dockerfile .
```

docker build 执行成功之后，我们再运行 docker images 这个命令，就可以看到生成的镜像了。

复制代码

```
1 # docker images
2 REPOSITORY  TAG   IMAGEID  CREATED   SIZE
3 registry/httpd  v1  c682fc3d4b9a  4 seconds ago  277MB
```

启动一个容器（Container）

做完一个镜像之后，你就可以用这个镜像来启动一个容器了，我们刚才做的镜像名字是 registry/httpd:v1，那么还是用 docker run 这个命令来启动容器。

复制代码

```
1 # docker run -d registry/httpd:v1
```

容器启动完成后，我们可以用 docker ps 命令来查看这个已经启动的容器：

复制代码

```
1 # docker ps
```

```
2 CONTAINER ID      IMAGE         COMMAND     CREATED       STATUS         PORTS
3 c5a9ff78d9c1        registry/httpd:v1   "/sbin/httpd -D FORE…"   2 seconds ago
```

在前面介绍 Dockerfile 的时候，我们说过做这个镜像是用来提供 HTTP 服务的，也就是让用户可以下载 file1、file2 这两个文件。

那怎样来验证我们建起来的容器是不是正常工作的呢？可以通过这两步来验证：

第一步，我们可以进入容器的运行空间，查看 httpd 服务是不是启动了，配置文件是不是正确的。

第二步，对于 HTTP 文件服务，如果我们能用 `curl` 命令下载文件，就可以证明这个容器提供了我们预期的 httpd 服务。

我们先来做第一步验证，我们可以运行 `docker exec` 这个命令进入容器的运行空间，至于什么是容器的运行空间，它的标准说法是容器的命名空间（Namespace），这个概念我们等会儿再做介绍。

进入容器运行空间之后，我们怎么确认 httpd 的服务进程已经在容器里启动了呢？

我们运行下面这个 `docker exec` 命令，也就是执行 `docker exec c5a9ff78d9c1 ps -ef` ，可以看到 httpd 的服务进程正在容器的空间中运行。

复制代码

```
1 # docker exec c5a9ff78d9c1 ps -ef
2
3 UID        PID  PPID  C STIME TTY          TIME CMD
4 root         1     0  0 01:59 ?        00:00:00 /sbin/httpd -D FOREGROUND
5 apache       6     1  0 01:59 ?        00:00:00 /sbin/httpd -D FOREGROUND
6 apache       7     1  0 01:59 ?        00:00:00 /sbin/httpd -D FOREGROUND
7 apache       8     1  0 01:59 ?        00:00:00 /sbin/httpd -D FOREGROUND
8 apache       9     1  0 01:59 ?        00:00:00 /sbin/httpd -D FOREGROUND
```

这里我解释一下，在这个 `docker exec` 后面紧跟着的 ID 表示容器的 ID，这个 ID 就是我们之前运行 `docker ps` 查看过那个容器，容器的 ID 值是 `c5a9ff78d9c1` 。在这个 ID 值的后面，就是我们要在容器空间里运行的 `ps -ef` 命令。

接下来我们再来确认一下，httpd 提供文件服务的目录中 file1 和 file2 文件是否存在。

我们同样可以用 `docker exec` 来查看一下容器的文件系统中，httpd 提供文件服务的目录 `/var/www/html` 是否有这两个文件。

很好，我们可以看到 file1、file2 这两个文件也都放在指定目录中了。

复制代码

```
# docker exec c5a9ff78d9c1 ls /var/www/html
file1
file2
```

到这里我们完成了第一步的验证，进入到容器的运行空间里，验证了 httpd 服务已经启动，配置文件也是正确的。

那下面我们要做第二步的验证，用 `curl` 命令来验证是否可以从容器的 httpd 服务里下载到文件。

如果要访问 httpd 服务，我们就需要知道这个容器的 IP 地址。容器的网络空间也是独立的，有一个它自己的 IP。我们还是可以用 `docker exec` 进入到容器的网络空间，查看一下这个容器的 IP。

运行下面的这条 `docker exec c5a9ff78d9c1 ip addr` 命令，我们可以看到容器里网络接口 eth0 上配置的 IP 是 `172.17.0.2` 。

这个 IP 目前只能在容器的宿主机上访问，在别的机器上目前是不能访问的。关于容器网络的知识，我们会在后面的课程里介绍。

复制代码

```
# docker exec c5a9ff78d9c1 ip addr

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defa

    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

    inet 127.0.0.1/8 scope host lo

```

```
       valid_lft forever preferred_lft forever

168: eth0@if169: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue stat

    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0

    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0

       valid_lft forever preferred_lft forever
```

好了，获取了 httpd 服务的 IP 地址之后，我们随便下载一个文件试试，比如选 file2。

我们在宿主机上运行 `curl` ，就可以下载这个文件了，操作如下。很好，文件下载成功了，这证明了我们这个提供 httpd 服务的容器正常运行了。

复制代码

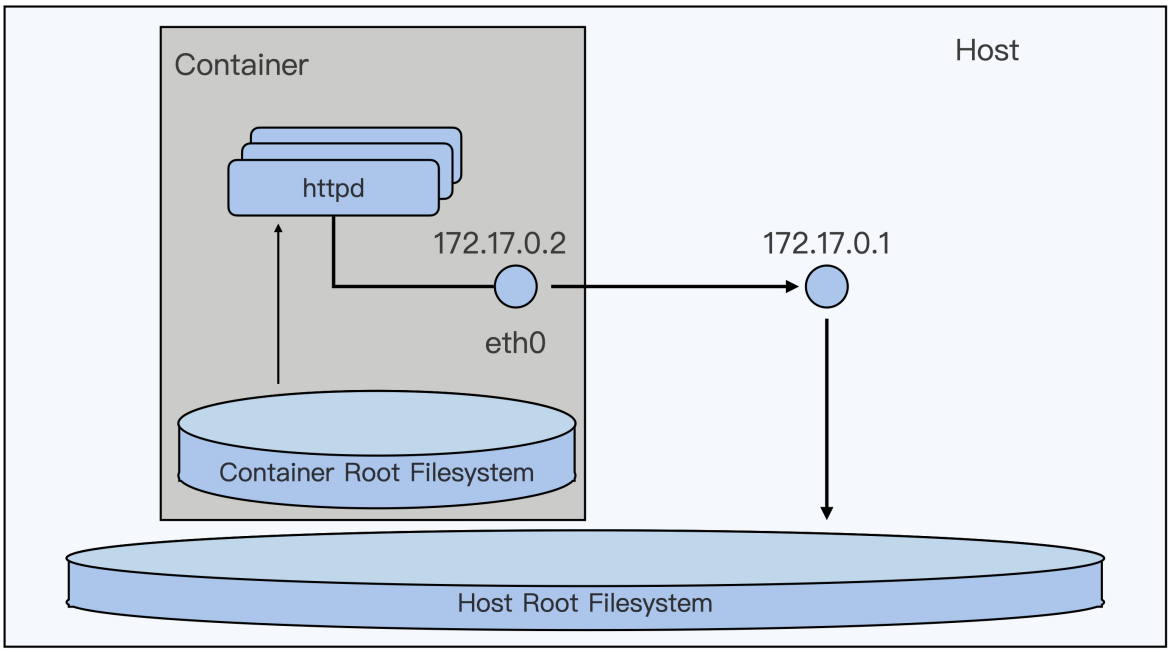
```
# curl -L -O http://172.17.0.2/file2
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Curren

                                 Dload  Upload   Total   Spent    Left  Speed

100     6  100     6    0     0   1500      0 --:--:-- --:--:-- --:--:--  1500

# ls
file2
```

上面的步骤完成之后，我们的第二步验证，用 `curl` 下载 httpd 服务提供的文件也成功了。

好了，我们刚才自己做了容器镜像，用这个镜像启动了容器，并且用 `docker exec` 命令检查了容器运行空间里的进程、文件和网络设置。

通过这上面的这些操作练习，估计你已经初步感知到，容器的文件系统是独立的，运行的进程环境是独立的，网络的设置也是独立的。但是它们和宿主机上的文件系统，进程环境以及网络感觉都已经分开了。

我想和你说，这个感觉没错，的确是这样。我们刚才启动的容器，已经从宿主机环境里被分隔出来了，就像下面这张图里的描述一样。



从用户使用的角度来看，容器和一台独立的机器或者虚拟机没有什么太大的区别，但是它和虚拟机相比，却没有各种复杂的硬件虚拟层，没有独立的 Linux 内核。

容器所有的进程调度，内存访问，文件的读写都直接跑在宿主机的内核之上，这是怎么做到的呢？

容器是什么

要回答这个问题，你可以先记住这两个术语 **Namespace 和 Cgroups**。如果有人问你 Linux 上的容器是什么，最简单直接的回答就是 Namesapce 和 Cgroups。Namespace 和 Cgroups 可以让程序在一个资源可控的独立（隔离）环境中运行，这个就是容器了。

我们现在已经发现：容器的进程、网络还有文件系统都是独立的。那问题来了，容器的独立运行环境到底是怎么创造的呢？这就要提到 Namespace 这个概念了。所以接下来，就先从我们已经有点感觉的 Namespace 开始分析。

Namespace

接着前面的例子，我们正好有了一个正在运行的容器，那我们就拿这个运行的容器来看看 Namespace 到底是什么？

在前面我们运行 `docker exec c5a9ff78d9c1 ps -ef`，看到了 5 个 httpd 进程，而且也只有这 5 个进程。

复制代码

```
# docker exec c5a9ff78d9c1 ps -ef

UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 01:59 ?        00:00:00 /sbin/httpd -D FOREGROUND
apache       6     1  0 01:59 ?        00:00:00 /sbin/httpd -D FOREGROUND
apache       7     1  0 01:59 ?        00:00:00 /sbin/httpd -D FOREGROUND
apache       8     1  0 01:59 ?        00:00:00 /sbin/httpd -D FOREGROUND
apache       9     1  0 01:59 ?        00:00:00 /sbin/httpd -D FOREGROUND
```

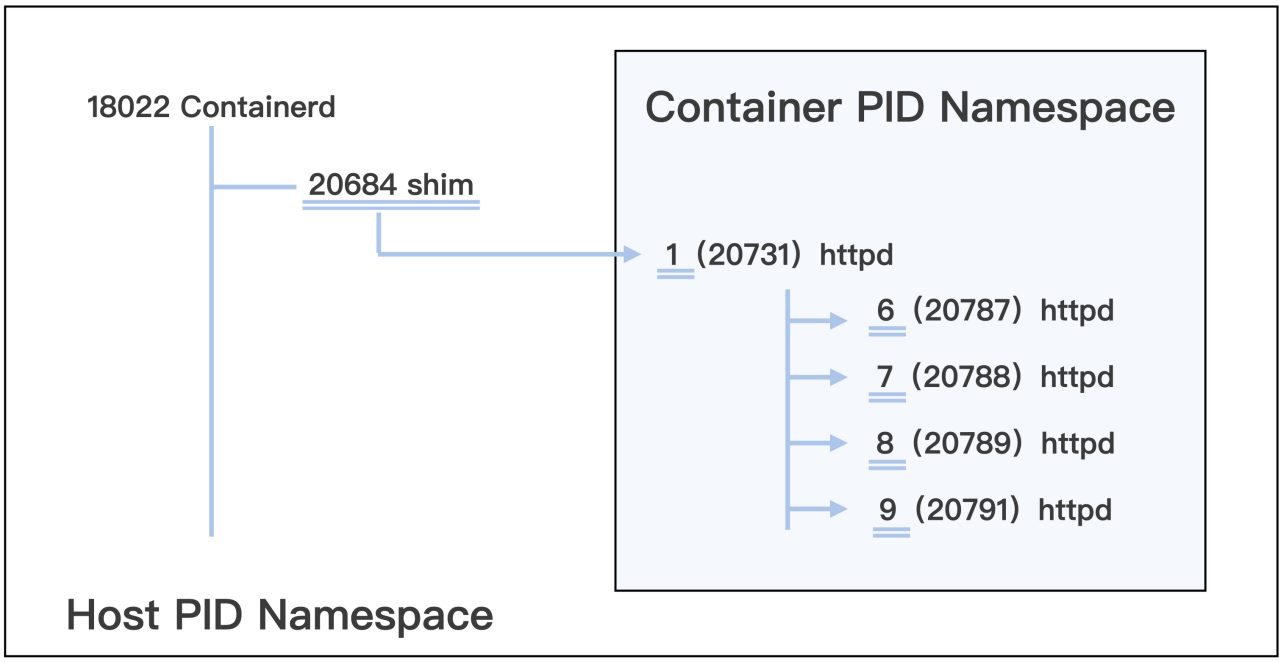
如果我们不用 `docker exec`，直接在宿主机上运行 ps -ef，就会看到很多进程。如果我们运行一下 `grep httpd`，同样可以看到这 5 个 httpd 的进程：

复制代码

```
# ps -ef | grep httpd

UID        PID  PPID  C STIME TTY          TIME CMD
root     20731 20684  0 18:59 ?        00:00:01 /sbin/httpd -D FOREGROUND
48       20787 20731  0 18:59 ?        00:00:00 /sbin/httpd -D FOREGROUND
48       20788 20731  0 18:59 ?        00:00:06 /sbin/httpd -D FOREGROUND
48       20789 20731  0 18:59 ?        00:00:05 /sbin/httpd -D FOREGROUND
48       20791 20731  0 18:59 ?        00:00:05 /sbin/httpd -D FOREGROUN
```

这两组输出结果到底有什么差别呢，你可以仔细做个对比，最大的不同就是**进程的 PID 不一样。**那为什么 PID 会不同呢？或者说，运行 `docker exec c5a9ff78d9c1 ps -ef` 和 `ps -ef` 实质的区别在哪里呢？

如果理解了 PID 为何不同，我们就能搞清楚 Linux Namespace 的概念了，为了方便后文的讲解，我们先用下面这张图来梳理一下我们看到的 PID。



Linux 在创建容器的时候，就会建出一个 PID Namespace，PID 其实就是进程的编号。这个 PID Namespace，就是指每建立出一个 Namespace，就会单独对进程进行 PID 编号，每个 Namespace 的 PID 编号都从 1 开始。

同时在这个 PID Namespace 中也只能看到 Namespace 中的进程，而且看不到其他 Namespace 里的进程。

这也就是说，如果有另外一个容器，那么它也有自己的一个 PID Namespace，而这两个 PID Namespace 之间是不能看到对方的进程的，这里就体现出了 Namespace 的作用：**相互隔离。**

而在宿主机上的 Host PID Namespace，它是其他 Namespace 的父亲 Namespace，可以看到在这台机器上的所有进程，不过进程 PID 编号不是 Container PID Namespace 里的编号了，而是把所有在宿主机运行的进程放在一起，再进行编号。

讲了 PID Namespace 之后，我们了解到 **Namespace 其实就是一种隔离机制，主要目的是隔离运行在同一个宿主机上的容器，让这些容器之间不能访问彼此的资源。**

这种隔离有两个作用：**第一是可以充分地利用系统的资源，也就是说在同一台宿主机上可以运行多个用户的容器；第二是保证了安全性，因为不同用户之间不能访问对方的资源。**

除了 PID Namespace，还有其他常见的 Namespace 类型，比如我们之前运行了 `docker exec c5a9ff78d9c1 ip addr` 这个命令去查看容器内部的 IP 地址，这里其实就是在查看 Network Namespace。

在 Network Namespace 中都有一套独立的网络接口比如这里的 lo，eth0，还有独立的 TCP/IP 的协议栈配置。

复制代码

```
# docker exec c5a9ff78d9c1 ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defa
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever

168: eth0@if169: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue stat
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

我们还可以运行 `docker exec c5a9ff78d9c1 ls/` 查看容器中的根文件系统（rootfs）。然后，你会发现，它和宿主机上的根文件系统也是不一样的。**容器中的根文件系统，其实就是我们做的镜像。**

那容器自己的根文件系统完全独立于宿主机上的根文件系统，这一点是怎么做到的呢？其实这里依靠的是 **Mount Namespace**，Mount Namespace 保证了每个容器都有自己独立的文件目录结构。

Namespace 的类型还有很多，我们查看"Linux Programmer’ s Manual"，可以看到 Linux 中所有的 Namespace：cgroup/ipc/network/mount/pid/time/user/uts。

在这里呢，你需要记住的是 **Namespace 是 Linux 中实现容器的两大技术之一，它最重要的作用是保证资源的隔离。**在后面的课程，讲解到具体问题时，我会不断地提到 Namespace 这个概念。

```
Namespace Flag            Page                   Isolates
Cgroup    CLONE_NEWCGROUP cgroup_namespaces(7)   Cgroup root directory
IPC       CLONE_NEWIPC    ipc_namespaces(7)      System V IPC,
                                                 POSIX message queues
Network   CLONE_NEWNET    network_namespaces(7)  Network devices,
                                                 stacks, ports, etc.
Mount     CLONE_NEWNS     mount_namespaces(7)    Mount points
PID       CLONE_NEWPID    pid_namespaces(7)      Process IDs
Time      CLONE_NEWTIME   time_namespaces(7)     Boot and monotonic
                                                 clocks
User      CLONE_NEWUSER   user_namespaces(7)     User and group IDs
UTS       CLONE_NEWUTS    uts_namespaces(7)      Hostname and NIS
                                                 domain name
```

好了，我们刚才说了 Namespace，这些 Namespace 尽管类型不同，其实都是为了隔离容器资源：**PID Namespace 负责隔离不同容器的进程，Network Namespace 又负责管理网络环境的隔离，Mount Namespace 管理文件系统的隔离。**

正是通过这些 Namespace，我们才隔离出一个容器，这里你也可以把它看作是一台“计算机”。

既然是一台“计算机”，你肯定会问这个“计算机”有多少 CPU，有多少 Memory 啊？那么 Linux 如何为这些“计算机”来定义 CPU，定义 Memory 的容量呢？

Cgroups

想要定义“计算机”各种容量大小，就涉及到支撑容器的第二个技术 **Cgroups （Control Groups）**了。Cgroups 可以对指定的进程做各种计算机资源的限制，比如限制 CPU 的使用率，内存使用量，IO 设备的流量等等。

Cgroups 究竟有什么好处呢？要知道，在 Cgroups 出现之前，任意一个进程都可以创建出成百上千个线程，可以轻易地消耗完一台计算机的所有 CPU 资源和内存资源。

但是有了 Cgroups 这个技术以后，我们就可以对一个进程或者一组进程的计算机资源的消耗进行限制了。

Cgroups 通过不同的子系统限制了不同的资源，每个子系统限制一种资源。每个子系统限制资源的方式都是类似的，就是把相关的一组进程分配到一个控制组里，然后通过**树结构**进行管理，每个控制组都设有自己的资源控制参数。

完整的 Cgroups 子系统的介绍，你可以查看 Linux Programmer’ s Manual 中 Cgroups 的定义。

这里呢，我们只需要了解几种比较常用的 Cgroups 子系统：

CPU 子系统，用来限制一个控制组（一组进程，你可以理解为一个容器里所有的进程）可使用的最大 CPU。

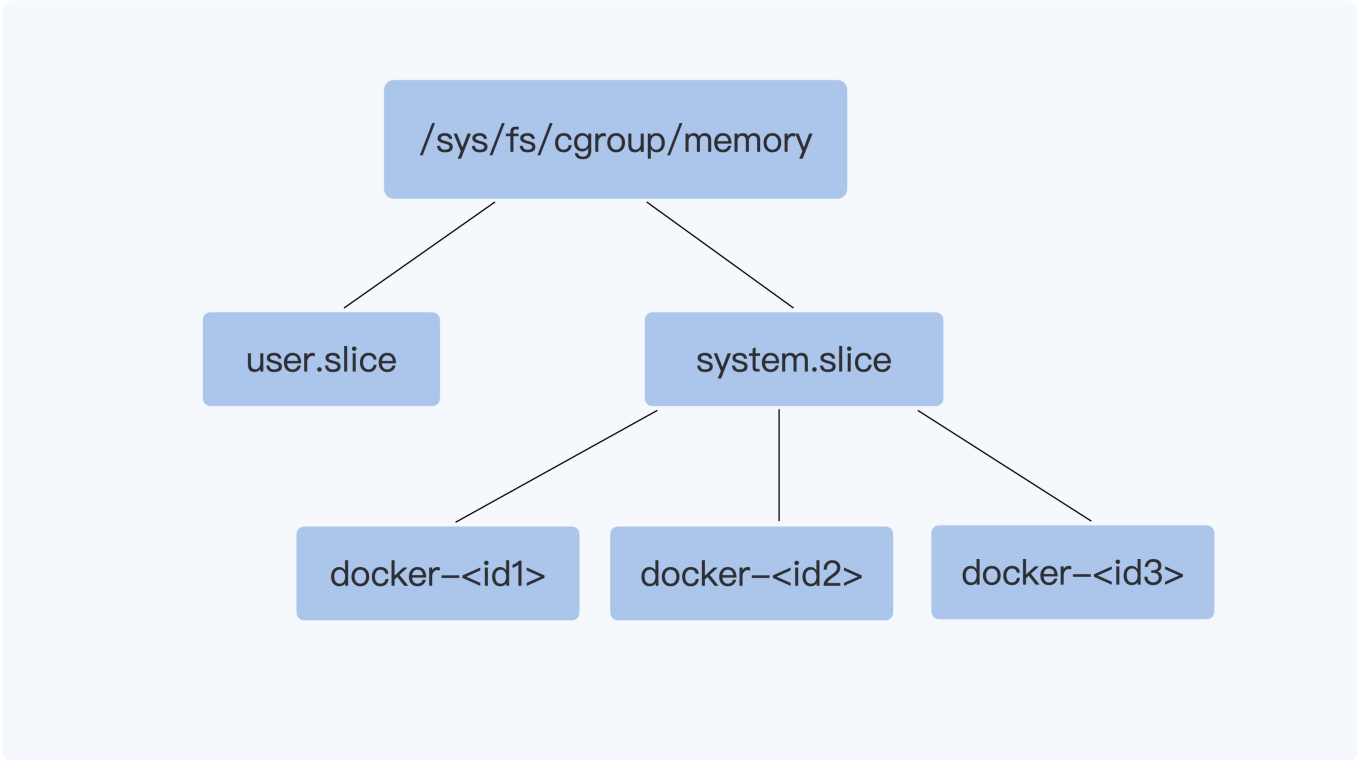
memory 子系统，用来限制一个控制组最大的内存使用量。

pids 子系统，用来限制一个控制组里最多可以运行多少个进程。

cpuset 子系统， 这个子系统来限制一个控制组里的进程可以在哪几个物理 CPU 上运行。

因为 memory 子系统的限制参数最简单，所以下面我们就用 memory 子系统为例，一起看看 Cgroups 是怎么对一个容器做资源限制的。

对于启动的每个容器，都会在 Cgroups 子系统下建立一个目录，在 Cgroups 中这个目录也被称作控制组，比如下图里的"docker-<id1>""docker-<id2>"等。然后我们设置这个控制组的参数，通过这个方式，来限制这个容器的内存资源。



还记得，我们之前用 Docker 创建的那个容器吗？在每个 Cgroups 子系统下，对应这个容器就会有一个目录 docker-**c5a9ff78d9c1……**这个容器的 ID 号，容器中所有的进程都会储存在这个控制组中 cgroup.procs 这个参数里。

你看下面的这些进程号是不是很熟悉呢？没错，它们就是前面我们用 ps 看到的进程号。

我们实际看一下这个例子里的 memory Cgroups，它可以控制 Memory 的使用量。比如说，我们将这个控制组 Memory 的最大用量设置为 2GB。

具体操作是这样的，我们把（2* 1024 * 1024 * 1024 = 2147483648）这个值，写入 memory Cgroup 控制组中的 memory.limit_in_bytes 里，**这样设置后，cgroup.procs 里面所有进程 Memory 使用量之和，最大也不会超过 2GB。**

复制代码

```
# cd /sys/fs/cgroup/memory/system.slice/docker-c5a9ff78d9c1fedd52511e18fdbd263


# cat cgroup.procs
20731
20787
20788
20789
20791

# echo 2147483648 > memory.limit_in_bytes
# cat memory.limit_in_bytes
2147483648
```

刚刚我们通过 memory Cgroups 定义了容器的 memory 可以使用的最大值。其他的子系统稍微复杂一些，但用法也和 memory 类似，我们在后面的课程中会结合具体的实例来详细解释其他的 Cgroups。

这里我们还要提一下 **Cgroups 有 v1 和 v2 两个版本**：

Cgroups v1 在 Linux 中很早就实现了，各种子系统比较独立，每个进程在各个 Cgroups 子系统中独立配置，可以属于不同的 group。

虽然这样比较灵活，但是也存在问题，会导致对**同一进程的资源协调比较困难**（比如 memory Cgroup 与 blkio Cgroup 之间就不能协作）。虽然 v1 有缺陷，但是在主流的生产环境中，大部分使用的还是 v1。

Cgroups v2 做了设计改进，**解决了 v1 的问题，使各个子系统可以协调统一地管理资源。**

不过 Cgroups v2 在生产环境的应用还很少，因为该版本很多子系统的实现需要较新版本的 Linux 内核，还有无论是主流的 Linux 发行版本还是容器云平台，比如 Kubernetes，对 v2 的支持也刚刚起步。

所以啊，我们在后面 Cgroups 的讲解里呢，主要还是用 **Cgroups v1 这个版本**，在磁盘 I/O 的这一章中，我们也会介绍一下 Cgroups v2。

好了，上面我们解读了 Namespace 和 Cgroups 两大技术，它们是 Linux 下实现容器的两个基石，后面课程中要讨论的容器相关问题，或多或少都和 Namespace 或者 Cgroups 相关，我们会结合具体问题做深入的分析。

目前呢，你只需要先记住这两个技术的作用，**Namespace 帮助容器来实现各种计算资源的隔离，Cgroups 主要限制的是容器能够使用的某种资源量。**

重点总结

这一讲，我们对容器有了一个大致的认识，包括它的“形”，**一些基本的容器操作**；还有它的“神”，也就是**容器实现的原理**。

启动容器的基本操作是这样的，首先用 Dockerfile 来建立一个容器的镜像，然后再用这个镜像来启动一个容器。

那启动了容器之后，怎么检验它是不是正常工作了呢?

我们可以运行 `docker exec` 这个命令进入容器的运行空间，查看进程是否启动，检查配置文件是否正确，检验我们设置的服务是否能够正常提供。

我们用 `docker exec` 命令查看了容器的进程，网络和文件系统，就能体会到容器的文件系统、运行的进程环境和网络的设置都是独立的，所以从用户使用的角度看，容器和一台

独立的机器或者虚拟机没有什么太大的区别。

最后，我们一起学习了 Namespace 和 Cgroups，它们是 Linux 的两大技术，用于实现容器的特性。

具体来说，**Namespace 帮助容器实现各种计算资源的隔离，Cgroups 主要对容器使用某种资源量的多少做一个限制。**

所以我们在这里可以直接记住：**容器其实就是 Namesapce+Cgroups。**

思考题

用 Dockerfile 为你最熟悉的应用程序做个镜像，然后用 Docker 命令启动这个容器。

欢迎在留言区分享你的疑惑和见解。你也可以把今天的内容分享给你的朋友，和他一起学习和进步。

13 人觉得很赞 | 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 开篇词 | 一个态度两个步骤，成为容器实战高手

下一篇 02 | 理解进程（1）：为什么我在容器中不能kill 1号进程?

精选留言 (24)

写留言

那时刻
2020-11-18

求教老师一个问题，我们有三个服务A B C部署在K8S里，它们各自设置了resouce limit(cpu, memory)，其中恰好 A1 B1 C1被分配到同一个node上，A2 B2分配到另外一个node

上。流量经过load balance之后打到 A1 A2和 B1 B2各自是均匀的，但是A1 B1使用的cpu 占比相对于 A2 B2要高出5% ~ 6%。我认为尽管docker通过namespace和cgroups来隔离和限制使用的资源，但是对于宿主机而言，各个docker实例之间还是存在竞争cpu...

展开

作者回复: @那时刻，

这种情况，如果A1/A2, B1/B2 都运行在自己的cgroup limit之内，那么需要先从两个宿主机的整体情况先做一个比较，从硬件类型开始，OS版本， 宿主机上的其他负载的情况，如果这些都一样，那么可以对A1/A2, B1/B2做perf看看热点函数。

3

朱雯

2020-11-17

老师好 1.我们现在对docker容器的使用就如同虚拟机一样 一个容器运行多个进程 但是我听说docker是天生是单进程的 是否需要改造 2.之前在某次创建docker容器时 要设置资源控制好像必须修改grub的某个配置 不配置就真的无法限制 这是啥原理 3. 容器使用过程中其实我大多数时候的交互方式是-it binbash的方式 当成虚拟机来用 这样是否有问题 4. 我们现在的环境 其实并没有做过多限制 容器之间相互干扰和影响 要设置限制 然后平滑的...

展开

作者回复: 1. 把容器当成虚拟机使用，可以是容器化过程中的一个过渡方案，但是长期的方向是native的方式来使用容器。

2. 我不清楚你具体修改的是哪个kernel参数，不过对于资源的限制比如cgroup, 或者xfs quota, 这些都是Linux 内核中的特性，所以有时候就需要在grub中对内核的启动参数做修改。

3. 这个没有什么问题。

4. 这个就需要用cgroup做限制了， 我在后面的课程里有很多这方面的介绍。

2 3

JianXu

2020-11-16

那张网络图里 172.17.0.21 是什么呢?

展开

作者回复: 这个是宿主机上Linux bridge interface docker0上的IP, 和容器里的IP在同一个网段，用来宿主机和容器进行网络通讯

2

JianXu
2020-11-16

” 会导致对同一进程的资源协调比较困难，比如 memory Cgroup 与 blkio Cgroup 之间就不能协作 “ --> CY 老师能展开介绍一下为什么 memory cgroup 和 blkio cgroup 不能协作的问题吗？为什么cgroup v2 就可以了呢?

另外，cgroup v1 可以管理disk IO 和 network IO 吗？那 cgroup v2 呢

展开

作者回复:
> 老师能展开介绍一下为什么 memory cgroup 和 blkio cgroup 不能协作的问题吗？为什么cgroup v2 就可以了呢?

这一部分我在第13讲"容器磁盘限速里" 会详细的解释。

cgroup v1 blkio 可以对 Direct I/O做限流。
cgroup v1 net_cls 配合 tc 可以对网络限流。

cgroup v2 io + memory 一起可以对 Direct I/O 和 Buffered I/O做限流，但是不支持xfs

2

conanforever22
2020-11-20

有个疑问, dockerfile中FROM后边的基础镜像应该怎么做?

作者回复: docker base image, 拿centos为例子，可以建一个目录，让后把需要的rpm 安装在这个目录下面，然后把目录打成一个tarball, 用docker import成为一个base image.

https://github.com/moby/moby/blob/master/contrib/mkimage-yum.sh

1 1

Marionxue
2020-11-17

使用了这么多年的Docker, 再回头看还是受益匪浅。

展开

1

上邪忘川
2020-11-16

centos7.4测试，docker在cgroup下的目录为/sys/fs/cgroup/memory/docker/40152218ccc9d2bcda0e146efdfed871a78480712cb9675404af800c2e812ea9/

作者回复: @上邪忘川,
请问一下, 你是否是通过systemd来启动dockerd的?

容器在cgroup下的目录树结构会受容器管理平台的影响。

2 1

heihei
2020-11-16

第一节就22分钟，我傻了，老师也太实在了😂

展开

1 1

Wen
2020-11-25

[root@localhost ~]# echo 2147483648 > /sys/fs/cgroup/memory/docker/7142bb204dd0d019696f8ac2111928898041e4a9e8c076e7bbadc1d3c032e21f/memory.limit_in_bytes
[root@localhost ~]# docker stats 7142bb204dd0
...

展开

作者回复: @Wen, 可以用 docker "--memory" 参数来启动容器。比如，
docker run --memory=400m -d registry/zombie-proc:v1

小白
2020-11-21

老师您好，我想请教一下，我想学习一下如何给各个服务合理分配 resouce limit，但不知道从哪下手，老师能指导一下关于这个知识的学习路径或者搜索关键词吗？谢谢。

作者回复: @小白
我想需要分析一下resouce有哪些， 可以先看Cgroups可以为哪些resouce做到哪种程度的limit。我在后面的课程里也会介绍cpu/memory/blkio/pids 这几个Cgroups.

exception
2020-11-20

老师好，我能否将windows开发环境打成一个Docker镜像，给新来的同事使用？如果可以的话，需要怎么操作。

展开

作者回复: Windows的容器我之前没有接触过，不过看到微软的网站有相关的介绍，你可以看一下。
https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/

JianXu
2020-11-19

关于 shim 进程

我在我的主机上：
$ ps -ef | grep httpd
root 18916 18885 0 17:30 ? 00:00:00 /usr/sbin/httpd -DFOREGROUND...

展开

作者回复: @JianXu,
你可以再检查一下，应该是有shim的。比如：

root 27204 18022 0 Nov17 ? 00:00:06 containerd-shim -namespace moby -workdir /var/lib/runtime/docker/containerd/daemon/io.containerd.runtime.v1.linux/moby/dfd5237b370c74aebfc706b849c665672f7de74f5210f00b887c4a0e7c31ae31 -address /var/run/docker/containerd/containerd.sock -containerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc -systemd-cgroup
root 27242 27204 0 Nov17 ? 00:00:12 /sbin/httpd -D FOREGROUND

JianXu
2020-11-19

我机器上的目录是 /sys/fs/cgroup/memory/system.slice/var-lib-docker-overlay2-c186a214dffd90302cdcbf43d1e749cfed5c4fa5d643e8ecd46408fbf6f4d39e-merged.mount$ 问题是我怎么去把 container id 和 这一串数字关联呢?

jianxu1@container:/sys/fs/cgroup/memory/system.slice/var-lib-docker-overlay2-...

展开

作者回复: @JianXu, 这是我第一次看到container在cgroup下的目录是用overlayfs merged dir id来命名的。需要看一下，你的机器OS/systemd/docker 版本，以及安装方式。

po
2020-11-18

老师，为什么我的 memory.limit_in_bytes 数字会这么大呢？我是rhel7.5。
[root@ocp3 docker-5950716c499d200837ce1c3061c2bf4ee8d1e33ce75b8844fe7c39c7b60e1671.scope]# cat memory.limit_in_bytes
9223372036854771712
[root@ocp3 docker-5950716c499d200837ce1c3061c2bf4ee8d1e33ce75b8844fe...

展开

作者回复: @po,
这个大值只是表示memory limit还没有设置

hadoop_admin
2020-11-18

发现照葫芦画瓢还不能执行成功。。。[root@testarm01 test]# docker build -t wqz/http:v1 -f ./Dockerfile
"docker build" requires exactly 1 argument.
See 'docker build --help'.
...

展开

作者回复: @hadoop_admin， 最后还有一个 “.”

docker build -t wqz/http:v1 -f ./Dockerfile .

Geek_35d340
2020-11-18

您好老师，请教一个问题，docker的IO限额只是针对direct io有效吗？生产中如果相对磁盘IO进行限制使用docker能满足需求吗?

展开

作者回复: 这个问题，需要先看一下你的Linux OS和docker的版本。
cgroup v1 只是支持 Direct I/O限流， cgroup v2可以支持 Buffered I/O.
不过目前，除了fedora 31+ 缺省支持cgroup v2外，其他的OS缺省的都是cgroup v1, 需要自己配置。docker/moby cgroup v2的代码应该已经合了，不过release出来的还不支持 v2。

生产中I/O限制，取决于你的应用了，如果应用是Direct I/O那么可以用docker

思维
2020-11-18

请教老师一个问题，执行 ls -l /proc/self/ns，我们的服务器上只支持6种namespace：ipc，mnt，net，pid，user，uts。请问这是怎么回事?

展开

作者回复: @思维，请看一下你的内核版本号。比如time namespace是Linux kernel 5.6之后才有的。

Helios
2020-11-18

请问老师，v1版本的资源协调能具体介绍下么，v2咋解决的呢，能说一下或者来个链接不~

作者回复: disk Buffered I/O 限流需要Cgroup v2, 我在后面的课程里会介绍到。

宋晓明
2020-11-17

老师容器命令审计有什么好的方案吗？ 因为我们目前在做k8s容器管理平台，有登录pod的需求，我只能捕获web socket流来记录命令，但平台之外的进入容器，我是无法捕获的

作者回复: @宋晓明
你说的登陆pod的方式是“kubectl exec”吗？ 还有“但平台之外的进入容器” 指的是什么方式?

1

chon
2020-11-17

希望在课程最后能给个书单，怎么才能掌握linux的内核，谢谢

作者回复: "Understanding the Linux Kernel"
"Linux Kernel Development"

虽然这两本书是10多年前的了，讲述的内核版本和目前的有差距，不过基本的概念原理都是一样的。认真读完，对于内核会有一个比较完整的认识。